ElMem: Towards an Elastic Memcached System

Ubaid Ullah Hafeez, Muhammad Wajahat, Anshul Gandhi; Stony Brook University
PACE Lab, Department of Computer Science, Stony Brook University
{uhafeez,mwajahat,anshul}@cs.stonybrook.edu

***Abstract*—Memory caches, such as Memcached, are a critical component of online applications as they help maintain low latencies by alleviating the load at the database. However, memory caches are expensive, both in terms of power and operating costs. It is thus important to dynamically scale such caches in response to workload variations. Unfortunately, stateful systems, such as Memcached, are not elastic in nature. The performance loss that follows a scaling action can severely impact latencies and lead to SLO violations.**

This paper proposes ElMem, an elastic Memcached system that mitigates post-scaling performance loss by proactively migration hot data between nodes. The key enabler of our work is an efficient algorithm, *FuseCache*, that migrates the optimal amount of hot data to minimize performance loss. Our experimental results on OpenStack, across several workload traces, show that ElMem elastically scales Memcached while reducing the post-scaling performance degradation by about 90%.

***Index Terms*—elastic, memcached, auto-scaling, data-migration**

I. Introduction

A crucial requirement for online applications is *elasticity* – the ability to add and remove servers in response to changes in workload demand, also referred to as autoscaling. In a physical deployment, elasticity can reduce energy costs by $\sim$ 30-40% [1]–[4]. Likewise, in a virtual (cloud) deployment, elasticity can reduce resource rental costs [5]–[8]; in fact, elasticity is often touted as one of the key motivating factors for cloud computing [9]–[11]. Elasticity is especially important for customer-facing services that experience significant variability in workload demand over time [12]–[15].

Customer-facing services and applications, including Facebook [12], [16] and YouTube [17], typically employ distributed memory caching systems, e.g., Memcached [18], to alleviate critical database load and mitigate tail latencies. While Memcached enables significant performance improvements, memory (DRAM) is an expensive resource, both in terms of power and cost. Our analysis of Memcached usage in Facebook [12] (see Section II) suggests that a cache node is *66% costlier* and consumes *47% more power* than an application or web tier node. Clearly, an elastic Memcached solution would be invaluable to customer-facing applications.

Unfortunately, memory caching systems are *not elastic* in nature owing to their statefulness. Consider a caching node that is being retired in response to low workload demand. All incoming requests for data items that were cached on the retiring node will now result in cache misses and increased load on the (slower) database, leading to *high tail latencies*.

Recent studies have shown that latencies and throughput degrade significantly, by as much as *10*$\times$, when autoscaling a caching tier [8]; worse, performance recovery can take *tens of minutes* [19]. Our own results confirm these findings as well (see Section II-D). Conversely, tail latencies for online services such as Amazon and Facebook are on the order of tens of milliseconds [20], [21]; even a subsecond delay can quickly translate to significant revenue loss due to customer abandonment.

Most of the prior work on autoscaling focus on stateless web or application tier nodes which do not store any data [4]–[7]. There is also some prior work on autoscaling replicated database tiers [2], [3]. Memory caching tiers are neither stateless nor replicated, and are thus not amenable to the above approaches. While there are some works that discuss autoscaling of Memcached nodes (see Section VI), they do not address the key challenge of performance loss following a scaling action.

This paper presents ElMem, an elastic Memcached system designed specifically to mitigate post-scaling performance loss. ElMem seamlessly migrates hot data items between Memcached nodes before scaling to realize the cost and energy benefits of elasticity. To minimize overhead, we implement ElMem in a decentralized manner and regulate data movement over the network (Section III).

The key enabler of ElMem is our novel cache merging algorithm, *FuseCache*, that determines the optimal subset of hottest items to move between retiring and retained nodes. *FuseCache* is based on the median-of-medians algorithm [22], and finds the n hottest items across any k sorted lists. We also show that *FuseCache* is within a factor $log(n)$ of the lower bound time complexity, $\mathcal{O}(k\ log(n))$ (Section IV).

We experimentally evaluate ElMem on a multi-tier, Memcached-backed, web application deployment using several workload traces, including those from Facebook [12] and Microsoft [23]. Our results show that ElMem significantly reduces tail response times, to the tune of 90%, and enables cost/energy savings by autoscaling Memcached (Section V). Further, compared to existing solutions, ElMem reduces tail response times by about 85%.

To summarize, we make the following contributions:

1) We present the design and implementation of ElMem, an elastic Memcached system.
2) We develop an optimal cache migration algorithm, *Fuse-*

Cache, that enables ElMem and runs in near-optimal time.
3) We implement ElMem and experimentally illustrate its benefits over existing solutions.

The rest of the paper is organized as follows. Section II provides the necessary background and motivation for our work. Section III describes the system design of ElMem and Section IV presents our *FuseCache* algorithm. We present our evaluation results in Section V. We discuss related work in Section VI and conclude in Section VII.

II. Background, Motivation, and Challenges

Online services are often provided by multi-tier deployments consisting of load-balanced web/application servers and data storage servers. To avoid performance loss due to slow I/O access at the data tier(s), many application owners employ memory caching systems. These systems provide low latency responses to clients *and* alleviate critical database load by caching hot data items on memory (DRAM); several caching servers can be employed in a distributed manner to provide scalable memory caching. A popular caching system that is employed by several companies, including Facebook [12], [16], Twitter [24], Wikipedia [25], and YouTube [17], is Memcached [18]. In the remainder of this paper, we focus on Memcached as our memory caching system.

A. Memcached overview

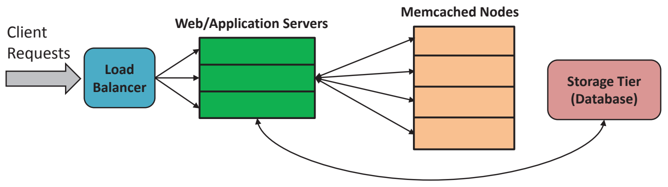


Fig. 1. Illustration of a multi-tier Memcached-backed application.

Memcached is a distributed in-memory key-value (KV) store that serves as an in-memory cache. Memcached sits in between the client and the back-end database or storage tier, as shown in Figure 1, and aggregates the available memory of all nodes in the caching tier to cache data. The memory allocated to Memcached on a node is internally divided into 1MB pages. The pages are grouped into *slabs*, where each slab is responsible for storing KV pairs, or items, of a given size range (to minimize fragmentation) by assigning each item to a *chunk* of memory. Within a slab, KV pairs are stored as a doubly-linked list in Most-Recently-Used (MRU) order.

Clients read (get) and write (set) KV data from Memcached via a client-side library, such as libmemcached [26]. The library hashes the requested key and determines which Memcached node is responsible for caching the associated KV pair; note that each key maps (via hashing) to *one* specific node. In case of a get, the KV pair is fetched from the faster (memory access) Memcached node. Else, the client library can decide to request the KV pair from the slower (disk access), persistent database tier, and optionally insert the retrieved pair into Memcached. Write requests proceed similarly; the client can choose to additionally write the KV pair to the database. *Consistent hashing* is typically employed to minimize the change in key membership upon node failures.

Note that the client library, and *not* Memcached, determines which node to contact. Memcached nodes are not aware of the key range that they (or the other nodes) are responsible for storing, thus placing this responsibility on the client. Each Memcached node can be treated as a simple cache which stores items in memory. If the number of items stored exceeds the memory capacity, items are evicted using the Least-Recently-Used (LRU) algorithm in $\mathcal{O}(1)$ time by simply deleting the tail of the MRU list.

B. Cost/Energy analysis of Memcached

Despite the many benefits of Memcached, it is an expensive solution, both in terms of cost and energy, because of its DRAM usage. Recent numbers from Facebook suggest that they use 72GB of memory per Memcached node [27]. By contrast, the web or application tier nodes are equipped with 12GB of memory. Memcached nodes typically have a Xeon CPU [16], and we expect application tier nodes to have about twice the computing power as that of a Memcached node. Using power numbers reported by Fan et al. [28], and normalizing them to get per GB and per CPU socket power consumption, we estimate that an application tier server (2 CPU sockets, 12GB) will consume 204 Watts of (peak) power, whereas a Memcached node (1 CPU socket, 72GB) will consume 299 Watts (*47% additional power*). In terms of cost, a compute-optimized EC2 instance currently costs \$0.1/hr, whereas a memory-optimized EC2 instance currently costs \$0.166/hr (*66% higher cost*), based on numbers for large sized instances [29].

C. Potential benefits of an elastic Memcached

Online services that employ Memcached often exhibit large variations in arrival rate [1], [14], [30], presenting an *opportunity* for reducing operating costs by dynamic scaling. For example, production traces from Facebook [12] indicate load variations on the order of $2\times$ due to the diurnal nature of customer-facing applications, and a further 2–$3\times$ due to traffic spikes. Our preliminary analysis of these traces reveals that a perfectly elastic Memcached tier—one that instantly adds/removes the optimal number of nodes and consolidates all hot data on the resulting nodes—can reduce the number of caching nodes by 30–70%. Unfortunately, dynamic scaling of Memcached is a *difficult* problem.

D. The inelastic nature of Memcached

Stateful systems store data that is required for the efficient functioning of the application. In the case of Memcached, hot data is cached in memory to mitigate load on the critical database tier. By design, stateful systems are not elastic due to their data dependence. Building elastic stateful systems thus requires careful consideration of the data on each node.

The key challenge in designing elastic stateful systems is the immediate, albeit transient, *performance degradation* after scaling. Addition of a new cache node results in a cold cache,

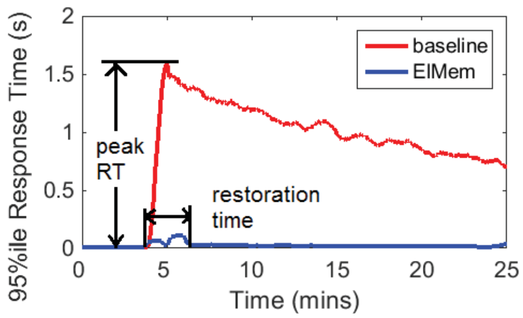


Fig. 2. Post-scaling performance degradation for Memcached.

whereas removal of an existing cache node results in loss of hot data. In both cases, performance can suffer severely due to cache misses—until the cache is warm again, which can take several minutes.

The red line (baseline) in Figure 2 shows the steep increase in 95%ile response time, from about 6ms to 1600ms, when Memcached is scaled in from 10 VMs to 9 VMs when using the Facebook ETC demand trace [12] (see Section V-A for details on our experimental setup). This significant increase in response time (RT), which we refer to as *peak RT* (shown in the figure), can hurt performance SLOs. Likewise, the time to revert to stable RTs, referred to as *restoration time*, dictates the duration of performance degradation; in Figure 2, the baseline's restoration time is more than 30 minutes. We refer to this overall loss in performance as ***post-scaling performance degradation***.

Most of the existing work on elastic stateful systems either ignores the post-scaling performance degradation problem (e.g., Amazon ElastiCache [31] ignores this crucial performance loss [32]) or assumes that data is replicated and thus at least one copy will exist (e.g., Sierra [3] and Rabbit [2]), which is not the case for Memcached. Our goal is to *address this critical gap in the design of stateful systems by specifically mitigating the post-scaling performance degradation*. The blue line (ElMem) in Figure 2 shows the improved performance under our approach, with peak RT reducing from 1600ms to 130ms and restoration time reducing from more than 30 minutes to about 2 minutes.

III. ElMem System Design

To enable an elastic Memcached design, ElMem specifically focuses on mitigating the post-scaling performance degradation. The design of ElMem is motivated by the observation that post-scaling degradation is caused by cache misses (due to a cold cache); thus, if we can identify and migrate the hot items prior to scaling, we can mitigate post-scaling degradation. Note that the hotness of an item refers to its recency of access; Memcached already stores the most recently used (MRU) access timestamp of each item.

The key challenge in mitigating post-scaling degradation lies in efficiently determining the correct subset of hot items to migrate between appropriate nodes. This section discusses the system design of ElMem, while our *FuseCache* algorithm that efficiently migrates hot items is presented in Section IV.

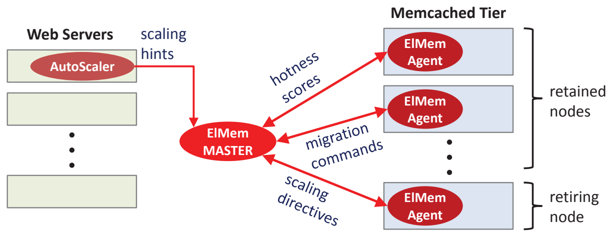


Fig. 3. ElMem's solution architecture, shaded in red. We only show components involved in the design of ElMem. The figure illustrates the case of one node being retired (scaled in).

We consider a multi-tier application deployment consisting of a Memcached tier comprised of several nodes, as shown in Figure 1. To dynamically scale Memcached while minimizing post-scaling performance degradation, the following questions must be addressed:

(Q1) *When and how much to scale?*
(Q2) *Which nodes to scale?*
(Q3) *How to migrate data prior to scaling?*

We are less concerned with the autoscaling policy that addresses Q1 (which is a pluggable module in our system design) and more concerned with how to minimize the post-scaling degradation when a scaling event is to be executed. Of the above three questions, Q3 and (to a lesser extent) Q2 are crucial to mitigate the post-scaling degradation. We first describe the solution architecture of ElMem, followed by our approach to address Q1, Q2, and Q3.

A. ElMem architecture

ElMem's solution architecture is shown in Figure 3. ElMem consists of a Master, an Agent on each Memcached node, and an AutoScaler on one of the web servers. In the case of scale in, we refer to nodes that are being turned off as *retiring* nodes, and those that remain as *retained* nodes, as shown in Figure 3. For scale out, we use the terms *new* nodes and *existing* nodes.

The AutoScaler monitors the keys requested from Memcached over time and uses this information to decide on autoscaling (Q1, see Section III-B); this decision is relayed as hints to the Master, who ultimately triggers the autoscaling. The AutoScaler can be located on any one of the web servers. Since incoming requests are load balanced among the web servers, sampling the keys at one web server allows us to infer the underlying popularity distribution of requested keys.

The Master is the lightweight central controller that orchestrates the autoscaling and the data migration prior to scaling, and can be located either on a separate node or colocated with a web server or the load balancer. In case of a scale-in decision, after receiving the autoscaling hints, the Master decides which nodes to scale based on a scoring mechanism that takes into account the hotness of items at each Memcached node (Q2, see Section III-C). Then, prior to executing the scaling, the Master initiates data migration between the retiring nodes and the retained nodes to mitigate the post-scaling performance loss (Q3, see Section III-D). Once the migration is complete, the Master informs the web servers about the change in Memcached composition and issues scaling directives to the retiring nodes to turn off. The scale-out case is similar, except that new nodes are first added, followed by data migration between existing nodes and new nodes, and only then does the Master inform the web servers about the change in Memcached composition.

The ElMem Agents (one at each Memcached node) communicate with each other, and the Master, to perform the actual data migration; this includes fetching the requested subset of KV pairs, transferring KV pairs to other nodes, and incorporating migrated KV pairs with locally cached pairs. The Agents are also responsible for inferring the hotness of items, as needed, for scaling and migration decisions.

B. When and how much to scale?

Dynamic scaling of the Memcached tier starts with addressing Q1, that is, when to scale and how much to scale by? To address Q1, we consider the maximum request rate, say r_{DB}, that the database or back-end storage tier can handle without violating the SLO performance target; here, we assume that database is the performance bottleneck, which is typically the case [8]. The AutoScaler then uses this estimate to determine, for a given incoming request rate, say r, the minimum Memcached hit rate, say p_{min}, to ensure that no more than r_{DB} req/s go to the database. Specifically:

$$r \cdot (1 - p_{min}) < r_{DB} \implies p_{min} > \left(1 - \frac{r_{DB}}{r}\right) \qquad (1)$$

As in prior work [2], [3], [33], [34], r_{DB} can be obtained by profiling the database or by examining database logs.

If the incoming request rate into the system, r, increases significantly, then we must add more Memcached nodes (scale-out) to satisfy p_{min} as per Eq. (1); note that r_{DB} is typically a constant for a given database configuration. On the other hand, if r decreases, then p_{min} will decrease (per Eq. (1)), possibly allowing us to save on costs by scaling in the size of the Memcached tier. r can be easily monitored online at the load balancer; Apache [35] and Nginx [36] load balancers already provide this ability.

To determine the need for, and amount of, Memcached scaling, the AutoScaler employs the *stack distance* measure to derive the memory capacity that achieves p_{min}. The stack distance of an item, say x, is defined as the number of unique items requested between successive requests to x. By tracking the stack distance of items over a trace of requests, we can determine the number of cache hits and misses for *all* cache sizes in a single pass over the trace [37]. ElMem employs this idea, by using the MIMIR [38] implementation to periodically compute the amount of memory required for every integer hit rate percentage (in a single pass) based on the request trace. The difference between required memory and current Memcached memory capacity, normalized by the memory capacity of each node, is used to determine the number of nodes to scale-in or scale-out. Since we cannot exactly predict future Memcached requests, we use the recent history of requests as our representative trace, similar to prior work on storage workload modeling [39], [40].

In summary, the AutoScaler periodically (every minute) employs Eq. (1) to derive p_{min} and then uses the stack distance measure over the recent trace of cache requests to determine the amount of scaling. Given the simple expression in Eq. (1) and the efficient implementation of the stack distance algorithm, the above computation takes less than a second. The AutoScaler then relays this information to the ElMem Master. In our implementation of ElMem, the exact autoscaling algorithm is a pluggable module. Thus, the user can input a different autoscaling algorithm, such as a predictive scaling framework [6], [41], if needed.

C. Which nodes to scale?

When scaling in, an important question is which Memcached nodes to turn off. Ideally, we want to migrate the hottest items from the retiring nodes to the retained nodes. Choosing a node that has very little hot data allows ElMem to quickly migrate this data to other nodes and scale in. On the other hand, a node that has a lot of hot data on it will require significant migration time before the scaling in event, resulting in lost opportunity costs.

Ideally, we should pick the node whose hot data migration requires the least amount of bytes to be transferred over the network. However, finding such a node entails determining, for *each* node, the subset of data on each slab that is hotter (with respect to MRU timestamp) than the corresponding data on that slab on *all* other nodes.

ElMem avoids this overhead by only comparing the hotness of the *median* items, in MRU order, across nodes. Specifically, for each slab b, ElMem Agents determine the median item in the MRU ordered linked list and send the item's MRU timestamp to the Master. The Master then compares the timestamps of all median items, across nodes, for each slab. The motivation behind this approach is as follows: assume we have only 1 slab of items on 2 nodes, each of size n items, and we want to scale in to 1 node. By choosing the node with the colder median to retire, we are guaranteed an upper bound of $n/2$ items to be moved [22]. On the other hand, if we randomly choose a node, we may have to move, in the worst case, all n items.

To account for the impact of different slabs, ElMem considers the weighted sum of slab scores, $s_{b,i}$, and the percentage of memory pages assigned to this slab, w_b. Thus, to retire a node, the Master chooses the node which is $\underset{i}{\text{argmin}}(\sum_b s_{b,i} \cdot w_b)$, where the summation is over all slabs. To retire x nodes, the Master chooses the x distinct values of i that result in the x smallest weighted sums. The choice of the node(s) is then relayed to all Agents on retiring nodes to execute the migration (as discussed in the next subsection). We show, in Section V, that this strategy results in the optimal node choice for scaling in almost all the traces we consider, and provides almost a 36% reduction in the number of items migrated compared to a random strategy.

D. How to migrate data prior to scaling?

The final and crucial step in autoscaling of Memcached is to address the post-scaling performance degradation. While our key component for addressing this issue is the *FuseCache* algorithm (presented in the next section), we first describe the system design for this component here. Most of our discussion is geared towards scale-in, though the design is similar for scale-out, and is discussed at the end of this subsection.

ElMem addresses post-scaling degradation by correctly identifying the subset of hot items on retiring nodes and migrating them efficiently to retained nodes, prior to scaling. Consider an item x belonging to a slab with chunk size b on a retiring node. Based on the design of Memcached, x must be migrated to a slab with chunk size b. The target node for migrating x is uniquely computed by taking its hash. Thus, to determine whether to migrate x or not, ElMem must compare x's MRU access timestamp with those of items on the target node's corresponding slab. Based on these comparisons, the retiring nodes send their hot data to retained nodes, who will then merge this data with their existing cached data, evicting older items as needed. To minimize overhead during migration, ElMem executes the migration in three successive phases, as discussed below.

1) Metadata transfer from retiring to retained nodes

To facilitate comparison, each Agent on a retiring node, once informed of the autoscaling decision by the Master, hashes its keys using consistent hashing and sends them along with their timestamps to the (hashed) target retained nodes. Note that the hashing function takes as input the member list of nodes, and so we use only the list of retained nodes when hashing for this phase; thus, the autoscaling decision from the Master is relayed to Agents *prior* to this phase. To minimize the overhead of transferring the metadata over the network, we investigate the use of ssh, scp, rsync. We find that it is best to create a tarball of the metadata (without compression) and pipe the output directly to the retiring node over ssh. Further, in this phase, we only transfer keys (which are usually small, about 10s of bytes in the case of Facebook [12]) and timestamps (10 bytes), and not values (100-1000 bytes [12]).

2) Hotness comparison on retained nodes

The Agent on each retained node must now determine, for every slab, the subset of keys from each retiring node that are hotter, in terms of MRU timestamp, than its existing data. While this is a challenging task, the problem is simplified by the observation that keys on each slab are implicitly stored in MRU order; thus, we only need to determine the number of keys per slab that we want to migrate from every retiring node. Nonetheless, a naive comparison of k lists of n items each requires $\mathcal{O}(n \cdot k \ log(n \cdot k))$ time. Our optimal *FuseCache* algorithm, discussed in Section IV, solves this problem in $\mathcal{O}(k \ (log(n))^2)$ time, which is a factor $log(n)$ more than the theoretical lower bound. The Agents on the retained nodes then inform the Master about the number of keys to migrate from each retiring node.

3) Data migration from retiring to retained nodes

The Master directs the retiring nodes to transfer the required number of KV pairs per slab, as determined by *FuseCache*, to the retained nodes. Agents on the retiring nodes pipe this data directly to the retained nodes. On the retained nodes, the Agents invoke a thread to write the migrated data into the local Memcached by prepending them to the start of the MRU list, thus evicting the colder items at the end of the MRU list of the retained node. For efficiency, we implement this new thread as part of the Memcached source code. Note that, by design of our *FuseCache* algorithm, the items being evicted are necessarily colder (in terms of MRU timestamp) than the KV pairs being migrated.

Once the Master receives an acknowledgement from all retiring node Agents, it informs the clients on the web servers about the change in Memcached membership (from all nodes to only retained nodes), and sends directives to retiring nodes to turn off. This completes the scale-in process. We show, in Section V, that this entire migration process of three phases requires about 2 minutes, for our setup.

4) Extension to scale out

The process for scale out is similar to, but simpler than, the above described 3-phase process for scale in. Once the autoscaling decision has been relayed to Agents, each existing node hashes its keys (based on the scaled-out membership of nodes) and determines the set of its KV pairs that hash to the new nodes. Under consistent hashing, if we scale out from k nodes to $(k+1)$ nodes, then only about $\frac{1}{(k+1)}$ of the KV pairs on each of the k existing nodes will hash to the new $(k+1)^{th}$ node [42]. As a result, the total KV pairs to be migrated to the new node, from all existing nodes, will typically be less than the capacity of the new node. Thus, ElMem simply migrates all hashed KV pairs and sets them on the Memcached of the new node. In the rare case that the KV pairs to be migrated require more memory than the new node's capacity, we can run our *FuseCache* algorithm to determine the top KV pairs and set them. Once the migrated data is set, clients on the web servers are informed about the scaled-out membership of Memcached.

IV. THE *FuseCache* ALGORITHM

We now present the *FuseCache* algorithm that determines the subset of keys to migrate to mitigate post-scaling performance degradation. *FuseCache* is invoked when there is a need to determine the hottest KV pairs across different sets of KV pairs from different nodes. Specifically, consider the case where we are scaling in $(k-1)$ nodes, and each of these nodes send their hashed keys and timestamps, for a given slab, to a retained node that has space for n items in that slab. *FuseCache* must now determine the top n keys across all k lists (including its own list of items on that slab), where $n >> k$, typically. Since keys in Memcached are stored in MRU order, *FuseCache*'s goal is to pick the *top n hottest items from k different sorted lists*.

A naive way of picking the hottest items is to merge all k

lists into one list of N items, where $N > n$ is the total number of items across all lists, and then sort them in $\mathcal{O}(N\ log(N))$ time. An arguably better algorithm, the k-way merge [43], iteratively pops the hottest item across all k lists N times, resulting in time complexity of $\mathcal{O}(N{\cdot}k)$. Since we only require the top $n < N$ items, we can achieve our goal in $\mathcal{O}(n{\cdot}k)$ time. We can further reduce this time to $\mathcal{O}(n\ log(k))$ by using heaps to determine the hottest element in each step [43]. By contrast, *FuseCache* achieves this goal in $\mathcal{O}(k\ (log(n))^2)$ time, which is much quicker since typically $n >> k$.

A. Algorithm design

We have k sorted (in MRU order) lists of timestamps, each of size s_i, for $i = 1, 2, \ldots, k$. Of these, $(k-1)$ belong to the retiring nodes and one, say the k^{th}, is the retained node. Let $s_k = n$. Since the $(k-1)$ retiring nodes only send a subset of their keys (those that hash to the retained node, see Section III-D), we have $s_i < n$ for $i < k$. Our algorithm should find the hottest set of n keys from across all k lists.

Our *FuseCache* algorithm is presented in Algorithm 1 and returns the number of hottest items to pick in MRU order, toPick[i], from each list, i. *FuseCache*'s key idea is to employ the median-of-medians (MOM) algorithm [22] recursively to discard cold items in each round until we are left with the hottest n items. We now provide a high-level description of our algorithm; refer to Figure 4 for illustration. Let N_j be the number of items across all lists at the start of round j. Note that, in the worst case, each list is initially of size n, thus $N_1 = n \cdot k$.

We first find the medians of each of the k sorted list. Next, we find the MOM, the median item in the median list. By construction, we are guaranteed that at least $1/4^{th}$ of the items are colder than the MOM [22]; this is denoted by the green bottom-right quadrant in Figure 4 (here, for illustration, the lists are arranged in decreasing order of hotness). Likewise, at least $1/4^{th}$ of the items are hotter than the MOM.

Next, we find the insertion point of the MOM in all other lists using binary search; insertion point of the MOM in the median list is itself. Note that the insertion point will be towards the end of the MRU list for the hotter lists and closer to the beginning of the MRU list for the colder lists, as shown in Figure 4. Let the set of items hotter than the MOM be X; note that $|X| > N_1/4$. If $|X| > n$, we discard the at least $N_1/4$ items that are colder than the MOM (including the green quadrant in Figure 4), reducing our search space of n hottest items to at most $N_2 = 3N_1/4$ and moving to round 2. Now consider the case of $|X| \leq n$. Since $|X| > N_1/4$, this case is only possible when $N_1 < 4n$. Thus, as long as $N_j \geq 4n$, we have $|X| > n$, allowing us to discard $1/4^{th}$ of the search space, on average, and move to round $(j+1)$, where we repeat the entire process.

Now consider the first such m such that $N_m < 4n$. For round m, again, if $|X| > n$, we can discard the cold items and reduce the search space to $3/4^{th}$. Thus, consider the remaining case of $|X| \leq n$. If $|X| = n$, we have found our hottest n items, the set X. If $|X| < n$, we retrieve the set X of items and recurse the entire process to find the remaining $n - |X|$ hottest items from the remaining $N_m - |X|$ items. Since $|X| > N_m/4$, we have again reduced the search space by at least $1/4^{th}$. We repeat the above process until we find all n hottest items.

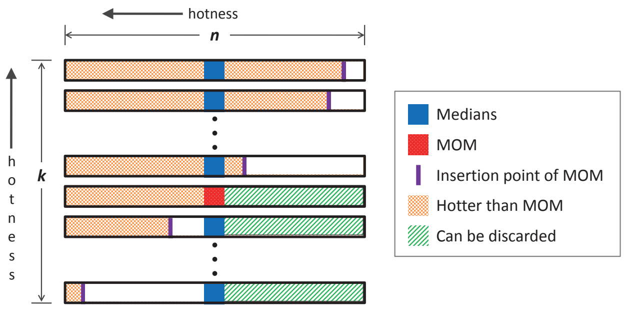


Fig. 4. Illustration of *FuseCache*. Leveraging the median-of-medians allows us to discard at least a quarter of the search space in initial rounds.

Algorithm 1 FuseCache

```
1: procedure FUSECACHE(k, [$list_1, list_2, \ldots, list_k$], n)
2:     $med \leftarrow [0_1, 0_2, ..., 0_k]$
3:     $startPt \leftarrow [0_1, 0_2, ..., 0_k]$
4:     $endPt \leftarrow [0_1, 0_2, ..., 0_k]$
5:     for $i \leftarrow 1 : k$ do
6:         $s[i] \leftarrow |list_i|$
7:         $startPt[i] \leftarrow 0$
8:         $endPt[i] \leftarrow s[i] - 1$
9:     while $n > 0$ do
10:        for $i \leftarrow 1 : k$ do
11:            $med[i] \leftarrow list[i][(startPt[i] + endPt[i])/2]$
12:        $MOM \leftarrow median(med, k)$
13:        $countX \leftarrow 0$
14:        $insertPts \leftarrow [0_1, 0_2, ..., 0_k]$
15:        for $i \leftarrow 1 : k$ do
16:            $size_i \leftarrow endPt[i] - startPt[i] + 1$
17:            $insertPts[i] \leftarrow$ $insertionPt(list[i], startPt[i], size_i, MOM)$
18:            $curCountX \leftarrow insertPts[i] + 1$
19:            $countX \leftarrow countX + curCountX$
20:        if $countX > n$ then
21:            for $i \leftarrow 1 : k$ do
22:                $endPt[i] \leftarrow startPt[i] + insertPts[i]$
23:        else if $countX \leq n$ then
24:            for $i \leftarrow 1 : k$ do
25:                $startPt[i] \leftarrow startPt[i]+insertPts[i]+1$
26:                $n \leftarrow n - countX$
27:    $toPick \leftarrow [0_1, 0_2, ..., 0_k]$
28:    for $i \leftarrow 1 : k$ do
29:        $toPick[i] \leftarrow endPts[i] + 1$
    return $toPick$
```

B. Time complexity

This algorithm runs in $\mathcal{O}(k\ (log(n))^2)$ time. For each list, finding the median takes $\mathcal{O}(1)$ time since lists are sorted. Finding the MOM from among the medians takes $\mathcal{O}(k)$ time. Finding the k insertion points via binary search takes $\mathcal{O}(k\ log(n))$ time. At each round, we reduce $1/4^{th}$ of the search space. Thus, to exhaust the initial (at most) $n \cdot k$ items, we will need $log(n \cdot k) = log(n) + log(k)$ rounds. Therefore, total time complexity is $\mathcal{O}(k\ (log(n))^2)$, considering $k < n$. This $\mathcal{O}(k\ (log(n))^2)$ complexity is significantly lower than the $\mathcal{O}(n\ log(k))$ complexity of k-way merge algorithms with heaps, especially for realistic Memcached deployments with hundreds or even thousands of nodes (k), with each node consisting of billions of items (n).

1) Theoretical lower bound on time complexity

The theoretical lower bound on time complexity for this problem is $\mathcal{O}(k\ log(n))$. To see this, note that our problem of determining the hottest n items can be reduced to the equivalent problem of picking an x_i for each of the k lists such that $\sum_{i=1}^{k} x_i = n$, where x_i is the number of top items, in MRU order, to pick from the i^{th} list to constitute the list of hottest n items. The number of possible solutions for all feasible x_i are $\binom{n+k-1}{n}$. Using a decision tree to solve this equivalent problem will require a tree with $\binom{n+k-1}{n}$ leaves, resulting in a height of $log\ \binom{n+k-1}{n}$. Thus, an optimal solution that makes the right decision at each level of the tree will require $\mathcal{O}(log\ \binom{n+k-1}{n})$ steps, which simplifies to $\mathcal{O}(k\ log(n))$.

V. Evaluation

A. Experimental Setup

For our evaluation, we set up a multi-tier Memcached-backed web application composed of several VMs deployed on an OpenStack cloud, similar to the one in Figure 1. At a high-level, the load generator creates PHP web requests and directs them at the load balancer, which in turn forwards the requests to Apache web servers (with PHP support). The web server parses the request and determines the data items needed to serve the web request; we fix the number of data items required per request to be 100 random KV pairs, whose popularity distribution can be controlled. The items are first requested from the Memcached tier via a multi-get (using the libmemcached library); note that several nodes might have to be contacted to serve all KV pairs. In case of a miss, the web server contacts the database. The fetched KV pairs from the database are inserted into Memcached, possibly leading to evictions. Note that the KV requests are get requests (read-only), thus no new KV pairs are written to the database.

We define *response time* (RT) for each web request to be the weighted average (over the 100 KV fetches) of the latencies of the get requests that hit in the Memcached and the remaining requests that are served by the database. We report tail RTs (95%ile RTs) when evaluating performance.

To generate load, we deploy httperf [45] on a large VM (8vCPU, 15GB RAM), and optimize it for high throughput. We use a single Apache web server VM (4vCPU, 8GB RAM), running mpm_prefork with mod_php, which employs the libmemcached library to communicate with Memcached. For the Memcached tier, we use a pool of 10 VMs, each with 2-vCPUs and 4GB memory, mimicking an economical cloud configuration; we use Memcached version 1.4.31. Finally, for the database, we employ ardb [46] (version 0.9.3), which uses the Redis protocol for communication and leverages RocksDB [47] as the back-end. The database runs on a physical machine with 8 cores and 32 GB RAM to mitigate the I/O bottleneck. Nonetheless, the bottleneck in our application is the database, which is typically the case in deployments [12]. Our database can handle a peak request rate of about 4,000 req/s before the latency rises abruptly; we thus set $r_{DB} = 4,000$ req/s when deciding on scaling (see Section III-B).

1) Modifications to Memcached

We add some custom functionality to Memcached to facilitate ElMem. First, we implement a timestamp dump command using LRU crawler routine in Memcached to write the MRU timestamps of a slab to a local file; this helps with the *FuseCache* algorithm (see Section III-D1). Second, we implement the batch import of KV pairs onto Memcached from a local file to help with *FuseCache* (see Section III-D3). This import functionality employs the set method of Memcached but removes the data checks since we already know the KV pairs are valid. Our modified Memcached source code is publicly available on Github [48].

2) Workload

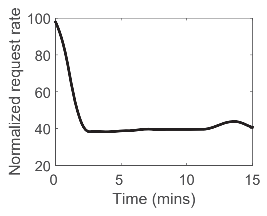
In terms of the workload, the key size is fixed at 11 bytes and the value sizes range from 1 byte to 1000 bytes. The value sizes follow a Generalized Pareto distribution with scale (σ) of 250.476 and shape ($\kappa = -\xi$) of 0.348238, similar to the distribution reported by Facebook [12]. The data set contains about 190 Million KV pairs, resulting in a size of about 60GB on the database. We use an exponential inter-arrival time distribution for the incoming requests, where the mean request rate changes dynamically as determined by the arrival trace (see below).

3) Traces

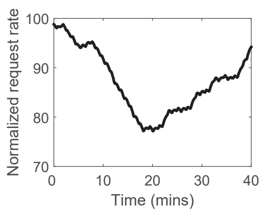
To drive our Memcached-backed multi-tier application, we use three different types of demand traces: (i) (digitized) Memcached traces from Facebook [12] – SYS and ETC, (ii) storage workload traces from Microsoft [23], and (iii) traces from online applications – SAP from an enterprise application [49] and NLANR from WITS [50]. Since our focus is on mitigating the post-scaling degradation, we consider trace snippets where demand varies considerably, as shown in Figure 5. We only show normalized values as these are modified per system capabilities.

B. Results

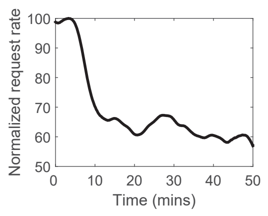
We first evaluate the benefits of ElMem's migration, including its choice of which node(s) to scale, in terms of



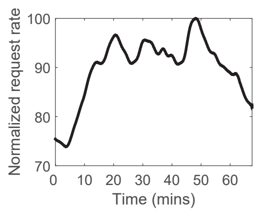
(a) SYS trace [12]



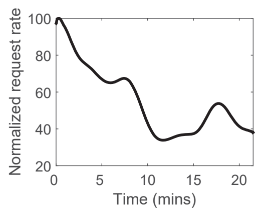
(b) ETC trace [12]



(c) SAP trace [41]

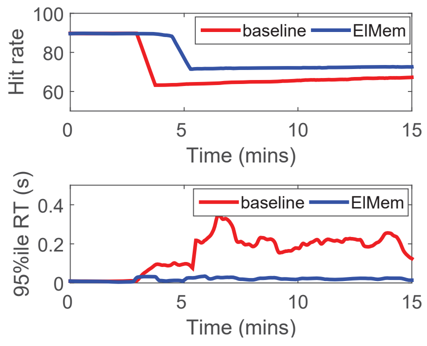


(d) NLANR trace [44]

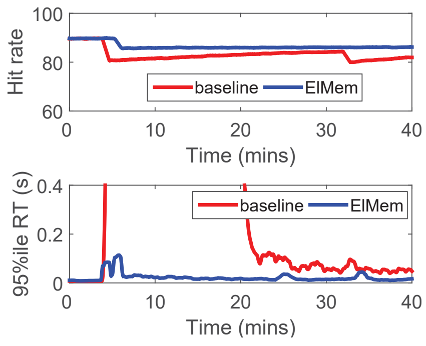


(e) Microsoft trace [23]

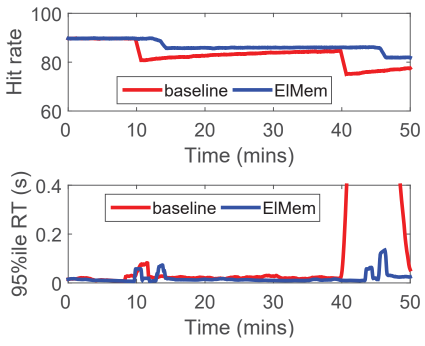
Fig. 5. Traces (normalized) used in our experiments.



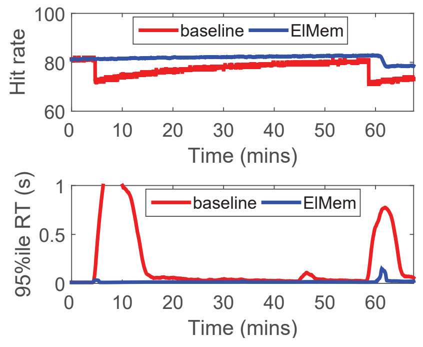
(a) SYS: 10 → 7 nodes.



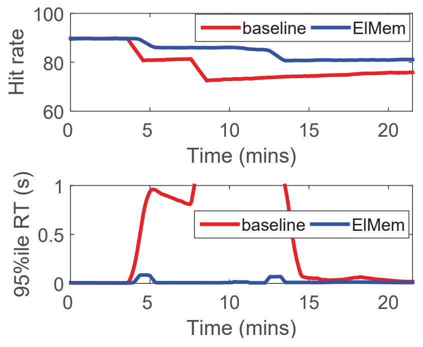
(b) ETC: 10 → 9 and 9 → 10 nodes.



(c) SAP: 10 → 9 and 9 → 8 nodes.



(d) NLANR: 8 → 9 and 9 → 8 nodes.



(e) Microsoft: 10 → 9 and 9 → 8 nodes.

Fig. 6. Experimental evaluation results illustrating the hit rate and 95%ile response time for ElMem and baseline for all traces. Numbers in the subcaption indicate the scaling action(s).

mitigating the performance degradation. Next, we evaluate *FuseCache*, including its overhead. We then evaluate the choice of which node to scale. Finally, we compare ElMem with other approaches.

1) Benefits of migration

The key evaluation for ElMem is the reduction in post-scaling performance degradation. For comparison, we consider a "baseline" approach that is informed about the autoscaling decision at the same time as ElMem, but it does not migrate items and immediately scales in or out, resulting in a cold cache. Thus, it uses the same approach for Q1 (Section III-B) and Q2 (Section III-C) as ElMem, but differs in the approach for Q3 (Section III-D).

Figure 6 shows the performance of baseline and ElMem for our traces. For each figure, the top graph shows the hit rate and the bottom graph shows the 95%ile response time, for each second. We start with Figure 6(a), which illustrates the performance for the SYS trace. Initially, the RT is about 5ms, which is in line with a high hit rate Memcached-backed application. At about the 3-min mark, the request rate drops significantly, translating to a scale in decision from 10 to 7 nodes, per the stack distance-based autoscaling scheme III-B. The baseline immediately autoscales from 10 to 7 nodes, without migration, resulting in the significant rise in RT from 5ms to 90ms, eventually peaking to 340ms; even after 10 minutes, the RT for baseline continues to be well above 100ms, a *20×* increase over the pre-scaling RT.

By contrast, ElMem migrates data and then scales down from 10 to 7 nodes at about the 5-min mark. After migration, the RT is much lower than the baseline; the peak RT under ElMem after the 3-min mark is about 35ms (compared to 340ms under baseline). In fact, the average of the 1-second 95%ile RTs after the 3-min mark reduces from 188ms under baseline to about 22ms under ElMem, a reduction of almost 88%. Note that the 2 minute difference for ElMem represents our migration and *FuseCache* overhead. Other results for scale down are similar, as in Figures 6(b) - 6(d), with an average *post-scaling performance degradation reduction of 96% for ETC, 90% for SAP, 92% for NLANR, and 97% for Microsoft*.

Results for performance improvement under scale out (for ETC and NLANR) are similar, with average post-scaling performance degradation reduction of 81%. Note that, immediately after scale out, we expect the hit rate under baseline to drop (due to cold cache) and that under ElMem to remain unchanged (due to *FuseCache* migration, which avoids cold cache). This is exactly what we observe for the scale out actions in Figures 6(b) and 6(d).

2) Overhead of FuseCache

For scale in and scale out, our *FuseCache* algorithm takes, on average, about 2 minutes. Specifically, the breakdown of *FuseCache*'s overhead is:

- about 2 seconds to score the nodes based on their medians (Section III-C),
- about 50 seconds to hash and dump data (Section III-D1),
- about 7 seconds to transfer the metadata (Section III-D1),
- less than 2 seconds to run *FuseCache*(Section III-D2),
- about 45 seconds to migrate the required data (Section III-D3), and
- about 8 seconds to set the migrated data into Memcached (Section III-D3).

We envision ElMem as being deployed in cases where the change in request rate is not intermittent, for example, diurnal changes in traffic or a sustained drop in request rate after a peak demand. In such cases, an overhead of about 2 minutes is not significant compared to, say, an hour of reduced request rate during which elasticity can help save substantial costs. Compared to baseline, which has no overhead in terms of delay in executing autoscaling, ElMem does have some overhead. However, as shown in Section V-B1, the baseline is an infeasible approach that significantly impacts tail response times. Thus, despite the small overhead of delay in autoscaling, ElMem is critical in realizing the design of an elastic Memcached system.

In terms of scalability of the overhead, we consider the time complexity of each of the steps outlined above. The scoring of the nodes takes $\mathcal{O}(s\ k)$ time, where s is the number of slabs per node and k is the number of nodes. *FuseCache* takes $\mathcal{O}(k\ (log(n))^2)$ time, as discussed in Section IV-B. The rest of the steps take at most $\mathcal{O}(n)$ time, where n is the number of items in each node; the exact complexity depends on the subset of hot items being migrated and is typically lower than $\mathcal{O}(n)$. Thus, the scoring step and *FuseCache* are linear in terms of number of nodes (k), while the other steps are linear in terms of the number of items being migrated, on average, per node.

3) Choice of which node to scale

During scale in, ElMem determines the node(s) to scale based on the hotness of their median items, as discussed in Section III-C. For a given scaling action, say scale in from 10 nodes to 9 nodes, the choice of node is only dependent upon the popularity distribution.

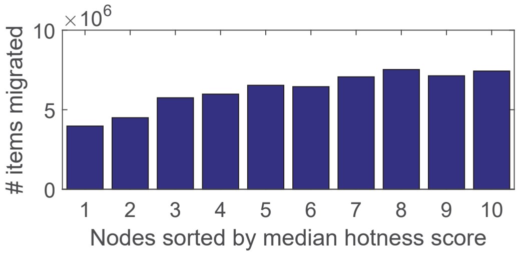


Fig. 7. Evaluation of node choice for scaling.

Figure 7 shows our experimental results for the number of items moved when scaling from 10 to 9 nodes depending on the choice of node for scaling; here, nodes are sorted in ascending order of median hotness score, as defined in Section III-C. ElMem picks the node with the coldest median score, that is, node 1; this results in migrating about 3.97 million items. By contrast, scaling a randomly selected node, which is typically the case in autoscaling [4], [6], results in migrating, on average, about 6.23 million nodes, an increase of about 60%. In the worst case, about 7.4 million items must be migrated, an increase of about 86%.

4) Comparison with other migration approaches

To further evaluate ElMem, we compare two other migration approaches:

1) Naive migrates $\frac{n-x}{n}$ fraction of items off of x randomly chosen nodes in response to a request to scale in x nodes. Naive migration assumes that the distribution of hotness of items within each node is similar. Thus, when scaling in from, say, n to $(n-1)$ nodes, Naive assumes that the coldest $1/n$ fraction of items of all nodes can be discarded.

2) CacheScale [8] migrates items from retiring to retained nodes based on the hotness of items inferred from incoming requests. Specifically, when informed about the autoscaling decision, CacheScale logically partitions the nodes into a primary cache and a secondary cache; the secondary cache is composed of retiring nodes. Incoming requests are first tried at the primary nodes; if they miss, they are retried at the secondary nodes. If the request hits in the secondary nodes, then it is migrated to the primary node, thus migrating hot items based on incoming request distribution. The secondary nodes are then discarded after some time; in our implementation of CacheScale, we discard them after about 2 minutes, which is the same as ElMem's overhead.

Figure 8 shows the performance of ElMem compared to that of Naive and CacheScale for a specific snippet of the SYS trace. In this case, the scaling decision (10 to 7 nodes) was made at about the 3-min mark. We see that the RT under ElMem is quite low, except for the roughly 1-min overhead during which RT is high. However, the RT under Naive and CacheScale continues to degrade well after the scaling event.

The performance under Naive is poor as it does not migrate the optimal set of hot items, and may in fact be evicting hot items when migrating colder items from other nodes.

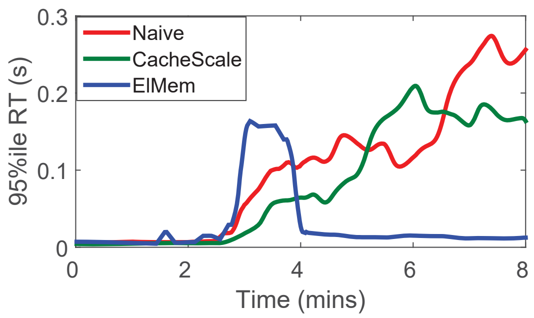


Fig. 8. Comparing ElMem's migration to other approaches.

For example, if Naive chooses the coldest node and migrates $\frac{n-1}{n}$ fraction of items from this node to a hot node, it may incorrectly evict hot items on that node. By contrast, ElMem would not migrate cold items. CacheScale also performs poorly as its migration depends on the hotness inferred based off of the incoming requests, which may not be accurate; further, the migration is dictated by the request rate and thus may be limited. Clearly, as seen in Figure 8, ElMem provides significant reduction in tail response times when compared to Naive (about 70% reduction) and CacheScale (about 64% reduction).

VI. Related Work

These is ample prior work on improving the performance of Memcached by addressing its network overhead (e.g., Chronos [51], Mcrouter [52], and Twemproxy [53]) and global locks (e.g., KV-Cache [54], MemC3 [55], and Mercury [56]). There is also some prior work on improving the performance of Memcached clusters by addressing hot spots or load imbalance (e.g., SPORE [57] and Zoolander [58]), communication overheads among nodes (e.g., AdaptCache [59]), and fault tolerance (e.g., Nishtala et al. [16]). Given the scope of our work, we focus on prior work related to dynamic scaling of Memcached.

Nishtala et al. [16] employ the "Cold Cluster Warmup" technique at Facebook when adding a new Memcached cluster by allowing clients to retrieve data (misses) from an existing warm Memcached cluster rather than persistent storage. However, this technique requires replicated Memcached clusters, and thus increases resource costs. CacheScale [8] proposes horizontal scaling of Memcached tiers by passively migrating data between Memcached nodes based on incoming requests. While effective, the restoration time of CacheScale critically depends on the arrival rate and popularity distribution, and can thus be arbitrarily high. By contrast, ElMem is independent of the arrival rate and popularity skew and is optimized, via the optimal *FuseCache* algorithm, to regulate the overhead of migration. Hwang et al. [60] propose an adaptive partitioning algorithm to re-balance the load created by hot items due to data skew. In a follow-up work [61], the authors discuss the challenges in designing self-managing caches and propose integrating dynamic scaling with load balancing, but do not discuss this further. Dynacache [62] is a cache controller that determines the best memory allocation and eviction policies for different applications in a shared Memcached-as-a-service setting. The authors later extended this work in Cliffhanger [63] to dynamically allocate memory among applications using a shared Memcached. However, this work incrementally adds more memory capacity (scale-up) to an existing node rather than adding new nodes (scale-out). We note that scale-up is not always feasible, especially at runtime, for physical deployments.

The stack distance concept is often used to efficiently determine the hit rate of caches and characterize them, as in MIMIR [38], Moirai [64], SHARD [65] and counter stack [66]. ElMem also employs stack distance, but specifically to facilitate scaling. Different from the above approaches, ElMem leverages stack distance to estimate the amount of memory needed to achieve a certain hit rate, and then translates this to scaling decisions and optimal migration.

There has also been recent work on alleviating the load imbalance, or hot spots, in Memcached caused by the uneven popularity of items. MBal [67] focuses on migrating and replicating items across Memcached worker threads (within and across nodes) to balance load. While the migration can be used to facilitate dynamic scaling as well, this aspect is not evaluated. SPORE [57] proposes a self-adapting, popularity-based data replication scheme to avoid hot spots. Zoolander [58] proposes using idle cache nodes for replication, and issuing requests to all replicas concurrently to avoid stragglers by using the first received result. While replication can be employed to improve performance, it requires *additional* memory resources and is thus contrary to our objective.

Lastly, there has been prior work on dynamic scaling of the database tier. Rabbit [2] and Sierra [3] propose elastic database tiers by organizing and scaling data *replicas* for storage systems such that at least one copy of the data is always on. Such approaches are not applicable to Memcached as it is not a persistent or replicated data store.

VII. Conclusion and Future Work

This work focuses on the critical post-scaling performance degradation problem that hinders the dynamic scaling of memory caches and other stateful clusters. We present the design of ElMem, an elastic system that dynamically scales Memcached in response to changes in workload demand. The key enabler of ElMem is our optimal *FuseCache* algorithm that finds the hottest items among Memcached nodes; further, *FuseCache*'s running time is within a logarithmic factor of the theoretical lower bound. Our experimental evaluation of ElMem, using workload traces from Facebook, Microsoft, and other web services, shows that we can substantially reduce the post-scaling degradation while accurately scaling Memcached; this elasticity translates to reduced energy consumption in physical deployments and reduced rental costs in virtual deployments.

Acknowledgements

This work was supported by NSF CNS grants 1617046, 1622832, 1717588, 1730128, and 1750109.

References

[1] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '08, San Francisco, CA, USA, 2008, pp. 337–350.

[2] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage," in *SOCC 2010*, Indianapolis, IN, USA, pp. 217–228.

[3] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: practical power-proportionality for data center storage," in *EuroSys 2011*, Salzburg, Austria, pp. 169–182.

[4] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. Kozuch, "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers," *Transactions on Computer Systems*, vol. 30, 2012.

[5] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *ICAC 2013*, San Jose, CA, USA, pp. 69–82.

[6] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in *CNSM 2010*, pp. 9–16.

[7] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *SOCC 2011*, pp. 5:1–5:14.

[8] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch, "Saving cash by using less cache." in *HotCloud*, 2012.

[9] Amazon Inc., "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2.

[10] Microsoft, "Azure Virtual Machines," https://azure.microsoft.com/en-us/services/virtual-machines/.

[11] Google Cloud Platform, "Cloud Compute Products," https://cloud.google.com/products/compute/.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *SIGMETRICS*, London, England, UK, 2012, pp. 53–64.

[13] B. Urgaonkar and A. Chandra, "Dynamic Provisioning of Multi-tier Internet Applications," in *ICAC 2005*, Seattle, WA, USA, pp. 217–228.

[14] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the 3rd ACM Symposium on Cloud Computing*, ser. SoCC '12, San Jose, CA, USA, 2012.

[15] L. A. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[16] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *NSDI 2013*, Lombard, IL, USA, pp. 385–398.

[17] C. Do, "YouTube Scalability," Seattle, WA, USA, 2007.

[18] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 2004, no. 124, pp. 5–5, Aug. 2004.

[19] Y. Cheng, A. Gupta, A. Povzner, and A. R. Butt, "High Performance In-memory Caching Through Flexible Fine-grained Services," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, Santa Clara, CA, USA, 2013.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, Stevenson, Washington, USA, 2007, pp. 205–220.

[21] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, San Jose, CA, USA, 2011, pp. 319–330.

[22] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time Bounds for Selection," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[23] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production Windows Servers," in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, Seattle, WA, USA, 2008, pp. 119–128.

[24] Twitter, "Twemcache: Twitter memcached," https://github.com/twitter/twemcache.

[25] mediawiki.org, "memcached," http://www.mediawiki.org/wiki/Memcached, 2014.

[26] "libMemcached," http://libmemcached.org/libMemcached.html.

[27] J. Taylor, "Capacity at Facebook," http://www.jedec.org/sites/default/files/Jason_Taylor_0.pdf, 2011.

[28] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, San Diego, CA, USA, 2007, pp. 13–23.

[29] "Amazon EC2 Pricing," http://aws.amazon.com/ec2/pricing.

[30] K. Wang, M. Lin, F. Ciucu, A. Wierman, and C. Lin, "Characterizing the Impact of the Workload on the Value of Dynamic Resizing in Data Centers," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12, London, England, UK, 2012, pp. 405–406.

[31] I. Amazon Web Services, "Amazon ElastiCache," http://aws.amazon.com/elasticache, 2014.

[32] H. Ganesan, "Deep dive into Amazon Elasticache: Elasticity Implications," http://harish11g.blogspot.in/2013/01/amazon-elasticache-memcached-internals_8.html.

[33] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *Proccedings of the 7th Conference on File and Storage Technologies*, ser. FAST '09, San Francisco, CA, USA, 2009, pp. 85–98.

[34] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal, "Pesto: Online Storage Performance Management in Virtualized Datacenters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11, Cascais, Portugal, 2011, pp. 19:1–19:14.

[35] The Apache Software Foundation, "Apache Module mod proxy balancer," http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html.

[36] I. Nginx, "nginx," http://nginx.org.

[37] G. Almási, C. Caşcaval, and D. A. Padua, "Calculating Stack Distances Efficiently," in *Proceedings of the 2002 Workshop on Memory System Performance*, ser. MSP '02, Berlin, Germany, 2002, pp. 37–43.

[38] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic Performance Profiling of Cloud Caches," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14, Seattle, WA, USA, 2014.

[39] T. Zhu, D. S. Berger, and M. Harchol-Balter, "SNC-Meister: Admitting More Tenants with Tail Latency SLOs," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16, Santa Clara, CA, USA, 2016, pp. 374–387.

[40] T. Zhu, M. A. Kozuch, and M. Harchol-Balter, "WorkloadCompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, Santa Clara, California, 2017, pp. 598–610.

[41] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah, "Minimizing Data Center SLA Violations and Power Consumption via Hybrid Resource Provisioning," in *Proceedings of the 2011 International Green Computing Conference*, ser. IGCC '11, Orlando, FL, USA, 2011, pp. 49–56.

[42] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. El Paso, TX, United States: ACM, 1997, pp. 654–663.

[43] D. Knuth, *The Art of Computer Programming*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1998, vol. 3, ch. 5.4.1.

[44] ITA, "http://ita.ee.lbl.gov/index.html," http://ita.ee.lbl.gov/index.html.

[45] D. Mosberger and T. Jin, "httperf—A Tool for Measuring Web Server Performance," *ACM Sigmetrics: Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[46] "ardb," https://github.com/yinqiwen/ardb.

[47] "RocksDB — A persistent key-value store," http://rocksdb.org/.

[48] U. U. Hafeez, "El Mem," https://github.com/PACELab/memcached_autoscale.

[49] SAP, "SAP application trace from anonymous source."

[50] WAND Network Research Group, "WITS: Waikato Internet Traffic Storage," http://www.wand.net.nz/wits/index.php.

[51] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable Low Latency for Data Center Applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, San Jose, CA, USA, 2012.

[52] Facebook, "Mcrouter," https://github.com/facebook/mcrouter.

[53] I. Twitter, "Twemproxy," https://github.com/twitter/twemproxy, 2012.

[54] D. Waddington, J. Colmenares, J. Kuang, and F. Song, “KV-Cache: A Scalable High-Performance Web-Object Cache for Manycore,” in *UCC 2013*, Dresden, Germany, pp. 123–130.

[55] B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing,” in *NSDI 2013*, Lombard, IL, USA, pp. 371–384.

[56] R. Gandhi, A. Gupta, A. Povzner, W. Belluomini, and T. Kaldewey, “Mercury: Bringing Efficiency to Key-value Stores,” in *Proceedings of the 6th International Systems and Storage Conference*, ser. SYSTOR ’13, Haifa, Israel, 2013.

[57] Y.-J. Hong and M. Thottethodi, “Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC ’13, Santa Clara, CA, USA, 2013.

[58] C. Stewart, A. Chakrabarti, and R. Griffith, “Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs,” in *Proceedings of the 10th International Conference on Autonomic Computing*, ser. ICAC ’13, San Jose, CA, USA, 2013, pp. 265–277.

[59] O. Asad and B. Kemme, “AdaptCache: Adaptive data partitioning and migration for distributed object caches,” in *Proceedings of the 17th International Middleware Conference*, Trento, Italy, 2016.

[60] J. Hwang and T. Wood, “Adaptive performance-aware distributed memory caching.” in *ICAC*, vol. 13, 2013, pp. 33–43.

[61] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. H. Huang, “Load Balancing of Heterogeneous Workloads in Memcached Clusters,” in *Feedback Computing*, 2014.

[62] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, “Dynacache: Dynamic Cloud Caching,” in *HotStorage*, 2015.

[63] A. Cidon, A. Eisenman, Daniel, M. Alizadeh, and S. Katti, “Cliffhanger: Scaling Performance Cliffs in Web Memory Caches,” in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, Santa Clara, CA, USA, 2016, pp. 379–392.

[64] I. Stefanovici, E. Thereska, G. O’Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, “Software-defined caching: Managing caches in multi-tenant data centers,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 174–181.

[65] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad, “Efficient MRC Construction with SHARDS,” in *Proceedings of the 2015 USENIX Conference on File and Storage Technologies*, 2015, pp. 95–110.

[66] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data, “Characterizing Storage Workloads with Counter Stacks,” in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, USA, 2014, pp. 335–349.

[67] Y. Cheng, A. Gupta, and A. R. Butt, “An in-memory object caching framework with adaptive load balancing,” in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.